Database - Feature #11884

Implement lazy transaction initialization for full transaction blocks

09/17/2026 06:18 AM - Artur Școlnic

Status:	Review	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Artur Școlnic	**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**reviewer:**	Alexandru Lungu, Ovidiu Maxiniuc
vendor_id:	GCD	**production:**	No
case_num:		**env_name:**	
version_reported:		**topics:**	
version_resolved:			

Description

History

#2 - 09/17/2026 06:18 AM - Artur Școlnic

- Status changed from New to WIP

- Assignee set to Artur Școlnic

#3 - 09/17/2026 06:48 AM - Artur Școlnic

Deferring transaction activation until DML is actually needed

1. Problem

A converted full-transaction block opens a real database transaction as soon as *any* record buffer for that database is touched — read or write. A great deal of converted code has the shape "open a full transaction, test a condition, write only if the condition holds", so the common path opens and commits a transaction that never wrote anything.

```
DO TRANSACTION:                       /* TxWrapper created, inactive    */
   FIND FIRST customer NO-LOCK.       /* openScope -> activate() -> BEGIN */
   IF customer.balance > 0 THEN       /* ...condition false...          */
      ASSIGN customer.flag = YES.
END.                                  /* COMMIT of a transaction that   */
                                      /* never wrote anything           */
```

The inactive-wrapper machinery already exists — TxWrapper.createTxWrapper creates wrappers that are not yet backed by a database transaction, and maybeActivateTxWrapper starts the real one later. Only the *trigger* is wrong. Today it fires from BufferManager.openScope (:2312), BufferManager.openScopeAt (:2658), RecordBuffer.initialize (:7778, :7923) and PersistenceInvocationHandler.invoke (:108).

The cost of a needless transaction is a BEGIN/COMMIT pair per block and — the expensive part on PostgreSQL — a snapshot held open for the whole duration of the block, holding back xmin and vacuum. Long read-only REPEAT loops are the worst case.

2. Proposed change

Move the activation trigger to **first record mutation**, per permanent database. The _temp database keeps the current buffer-scope trigger.

Two hooks, because one point is not sufficient:

- **Primary (early).** At the point a DMO transitions to "will be written" — record create, record delete, and the first field assignment — activate the wrapper for that DMO's database. This must run *before* BaseRecord's undoable-tracking call to Session.trackUndoable (BaseRecord.java:1849), because SavepointManager.trackUndoable (:485) asserts that the savepoint manager is already bound to a session.
- **Backstop (late).** In Persister — the single class that issues INSERT/UPDATE/DELETE SQL (insert, bulkInsert, bulkCopy, delete x2, update) — activate if not already active, and log when this path fires. The backstop firing means the primary hook missed a case; the log is how those cases are found, rather than shipping a silent write outside the transaction.

PersistenceInvocationHandler.invoke (the unsafe / direct-SQL proxy) keeps its unconditional activation — it cannot distinguish reads from writes.

3. Record lock bookkeeping must be decoupled from activation

Record locks are FWD-managed (LockManager), not database row locks. RecordLockContext.Perm defers release and downgrade while bufferManager.isTransaction() is true (RecordLockContext.java:504, :515), and the deferred release happens in RecordLockContext.transactionEnded() — called **only** from TxWrapper.end(), which TxWrapper.finished() skips for an inactive wrapper (TxWrapper.java:271-281).

Today that is unreachable: acquiring a lock requires a buffer, and opening a buffer scope activates the wrapper. After this change it becomes reachable — FIND ... EXCLUSIVE-LOCK with no subsequent write leaves the wrapper inactive and leaks the lock past the end of the transaction.

Proposed fix: split end() into a database half and a bookkeeping half. The bookkeeping half — RecordLockContext.transactionEnded() and the reclaimPendingKeys call that consumes its result — runs unconditionally in finished(), iterate() and retry(), whether or not the wrapper was ever activated. iterate() and retry() currently return early when inactive (TxWrapper.java:205-208, :227-230) and need the same treatment, otherwise an iterating read-only transaction block leaks locks at every iteration boundary.

4. Re-arming the deferral on iteration

TxWrapper.iterate() currently does end(); begin();. Once a REPEAT loop activates on one iteration, every later iteration carries a database transaction whether or not it writes. Since REPEAT with conditional DML is precisely the pattern under discussion, iterate() and retry() should instead **deactivate**: end the transaction, clear persistenceCtx, and return the wrapper to inactiveTxWrappers so the next iteration defers again. Session.commit already discards the savepoint manager, so re-installation on the next activate() is clean.

5. What does not change

TRANSACTION, _Trans, the TransactionManager transaction level and nesting, variable and temp-table UNDO, DirtyShareContext publication (there is nothing to publish without a mutation), and sequences (nextval is non-transactional on PostgreSQL regardless).

Reads in the pre-activation prefix run outside the transaction. Under READ COMMITTED that is a per-statement snapshot either way, so there is no visible difference; and an FWD session that intends to write holds an FWD exclusive lock, which already serialises FWD writers across the read-to-write window.

6. Configuration

persistence/defer-transaction-until-dml in the directory, read at bootstrap through Utils.getDirectoryNodeBoolean alongside force-no-undo-temp-tables (DatabaseManager.java:1789), **defaulting to true**. Setting it false restores the buffer-scope trigger, so a site that hits a regression reverts by configuration rather than by build.

7. Testing

A testcases battery covering: a read-only transaction block issues no BEGIN; conditional DML issues nothing on the false path and commits correctly on the true path; FIND ... EXCLUSIVE-LOCK with no write releases its lock at transaction end (the section 3 regression); nested sub-transaction blocks where the write happens in the innermost block; REPEAT writing on alternate iterations; UNDO both before and after the first DML; and multi-database blocks where only one database is written. Transaction counts asserted from TxWrapper instrumentation, lock state asserted cross-session through the two-session harness, and OpenEdge baselines captured for anything where 4GL semantics are in question.TxWrapper instrumentation, lock state asserted cross-session through the two-session harness, and OpenEdge baselines captured for anything where 4GL semantics are in question.

Risks

1. **Silent write outside a transaction.** This is the severe one. If a write path bypasses both the primary hook and the Persister backstop, it executes under autocommit and is committed immediately. UNDO then cannot roll it back and the failure is silent — no exception, just data that should not be there. The backstop and its logging exist specifically to bound this risk, but the risk is only as good as the enumeration of write paths.
2. **Record lock leak past transaction end.** Described in section 3. It is a genuine new regression introduced by the change, not a pre-existing one, and it is reachable from ordinary code (FIND ... EXCLUSIVE-LOCK with no write).
3. **Ad-hoc transaction collision.** Persistence.Context.beginTransaction returns false when a transaction is already open on the session, and TxWrapper.begin() throws IllegalStateException in that case unless initialTx is set. Deferring activation moves the window in which an ad-hoc transaction (for example the TEMP_CTX transactions in TemporaryBuffer, or a trigger touching another database) can be open when the master wrapper finally activates.
4. **Error timing shift.** A connection or transaction failure now surfaces at the first DML rather than at block entry. Code that expects to fail early — and any ON ERROR phrase scoped to catch it — sees the error at a different point in the block.
5. **Read-then-write staleness against non-FWD writers.** A record read before activation and updated after it was read outside the transaction. FWD's own lock manager serialises FWD sessions, but an external writer against the same database has a wider window than before.
6. **Savepoint manager re-installation.** Section 4 deactivates and reactivates wrappers across iterations. Session.beginTransaction throws if a savepoint manager is already installed, so the deactivation path has to leave the session genuinely clean; getting this subtly wrong produces an IllegalStateException only on loops that write on some iterations and not others.
7. **Reduced observability.** Anything that samples database transaction state — VSTs, monitoring, statement logs, support diagnostics — will report substantially fewer transactions. Existing dashboards and any tooling that infers activity from transaction counts will need to be re-read.

Open questions

1. Should section 4 (re-arming the deferral per iteration) land in this branch, or ship separately after sections 2, 3 and 6 have soaked? It carries the savepoint-manager risk above and is separable.
2. Is there any reason an EXCLUSIVE-LOCK read with no mutation needs a database transaction that I have not found? The lock manager is in-memory, but a dialect-specific escalation path would invalidate the premise.
3. Should the Persister backstop log, or fail fast in non-production builds? Failing fast finds missed write paths during testing; logging avoids turning a missed path into an abend at a customer.
4. Is the enumeration of write paths in section 2 complete — specifically, do audit record generation, database write triggers, and bulk/dynamic temp-table operations reach Persister, or do any of them reach the database another way?

Alex, please glance over the design and comment on the risks and open questions, thanks.

#4 - 09/17/2026 09:34 AM - Alexandru Lungu

I am not sure whether Session.beginTransaction is something to be deferred at all. My point is that there is no BEGIN TRANSACTION SQL being run to the database, so starting a transaction even though no changes will be done is a no-op. My big concern is with the actual commit that is an actual SQL. I think delaying the transaction begin would be more complicated than using a flag for the first actual change done in that bracket.

I think the design is targeting an upper layer, including record locks iteration, savepoint manager. I don't think that any of these are relevant for the cause of simply avoiding the COMMIT SQL to save performance. Managing this at a higher level can help us allocate less memory resources (TxWrapper instances and so on), but the optimization will be marginal.

I would like to scope this to the orm package and don't let the user (i.e. 4GL compatibility layer) plug into it.

A phase 1 of this task will be to abstract any JDBC connection through session if possible. Thus, Session will always know the database state at a moment of time. Arbitrary SQL run by direct-Java access is accounted as "drop any optimization chances".

A phase 2 would be to hook in the right API of the Session to detect if the ORM session actually changed the DB in any way, in order to make commit a no-op eventually.

#5 - 09/18/2026 09:43 AM - Artur Școlnic

- Status changed from WIP to Review

- reviewer Alexandru Lungu, Ovidiu Maxiniuc added

Summary of the changes

The change defers the SQL transaction until the ORM session actually writes, so a read-only transaction issues no commit at all.

Only the **database** transaction is affected. The change is confined to com.goldencode.p2j.persist.orm; TransactionManager, BlockManager, TxWrapper, BufferManager and RecordLockContext are untouched, so 4GL transaction scoping, the TRANSACTION function, _Trans and record-lock lifetimes are unchanged by construction.

Mechanism

Session now distinguishes two states: inTx (a transaction has been requested — unchanged semantics, still what inTransaction() returns) and txMaterialized (the JDBC connection has actually left auto-commit).

1. beginTransaction no longer clears auto-commit; it only sets the logical flag.
2. clearAutoCommit is the single point where the connection leaves auto-commit. It is driven from checkTx (which every DML entry point already called), setSavepoint, getConnection, and the new prepareForStreaming.
3. commit / rollback skip the SQL entirely when nothing was ever materialized. This is the saving.
4. Auto-commit is restored when a transaction ends, so the next transaction on that session can defer in its turn — without this the connection stays in transaction mode for the rest of the session's life and every later commit is genuinely owed.

Two connection accessors now exist: getConnection() materializes (the safe default for callers whose intent is unknown), and getConnectionForRead() does not. The hot read paths — record load, extent hydration, unique/PK probes, and the non-streaming query paths — use the read accessor; everything else keeps the materializing one.

Gated by persistence/defer-transaction-until-dml in the directory, default on.

Worth knowing

The saving is narrower than the ~550k commits/run measured in #10275 suggest. Savepoint creation materializes at the first **in-memory** DMO change, and scrolled queries must materialize to keep a server-side cursor, so elision only survives for transactions whose reads are all FIND / record hydration and which assign nothing. FOR EACH and OPEN QUERY do not qualify.

The subtle areas, all of which came out of review and are the places worth the most attention:

- Auto-commit is restored on the **success** path only — setAutoCommit(true) commits a transaction in progress, so restoring it after a failed

rollback would commit the very work the caller was told was discarded.
- prepareForStreaming clears auto-commit **even with no transaction open**, because a JDBC fetch size only yields a server-side cursor outside auto-commit and the common case for a large read is no transaction at all.
- beginTransaction clears auto-commit before mutating any session state, so a failure leaves the session exactly as it was.
- MetadataManager.removeDynamicTable now begins its transaction before taking the connection; the previous order left its deletes outside the transaction.

Testing

A new slice, tests/persistence/lazy_transaction — 125 ABLUnit tests in 11 classes. The patterns covered:

- read-only transaction blocks, and the conditional-DML pattern this work targets, on both branches;
- every distinct form of "first write" — assign, create, delete, validation-forced flush, write trigger, buffer-copy, dynamic buffer, sequence draw, and a write issued from a callee;
- reading-query shapes, both read-only and reading back the session's own uncommitted write, since query shape now determines whether a transaction materializes;
- iterating transaction blocks that write on only some iterations;
- transactions spanning internal, external and persistent procedure calls;
- error and rollback paths either side of the first write, including constraint violations raised by the flush itself, RETRY, labelled UNDO and CATCH;
- record locking with no write, and lock upgrade;
- temp-table and permanent-table writes in one block;
- the TRANSACTION function sampled across the materialization point;
- long sequences of alternating read-only and writing transactions on one session.

Results

Run	Pass	Fail
OpenEdge 12.8	**125**	**0**
FWD trunk (16758)	118	7
FWD with change, deferral on	119	6
FWD with change, deferral off	119	6

Ovidiu and Alex, please look over the changes so far before we move further with the design, the code is in 11884a.

#6 - 09/21/2026 02:56 AM - Artur Școlnic

- Status changed from Review to WIP

Will do a few more rounds of self review and generate more tests.

#7 - 09/21/2026 09:49 AM - Artur Școlnic

- Status changed from WIP to Review

The following issues have been solved:
Restrict prepareForStreaming to the only case that benefits from it — a forward-only cursor on a deferral-enabled session — so it no longer flips auto-commit on a temp database's shared connection or defeats the configuration switch, and correct the two javadoc comments that understated what the read accessor and the streaming residue actually do.

Tests committed to tests/persistence/lazy_transaction.

Please review rev 16764.